



08. Punteros y memoria dinámica

Programación - 1º DAM

Luis del Moral Martínez

versión 20.10

Bajo licencia CC BY-NC-SA 4.0



Contenidos del tema

1. Punteros
2. Memoria dinámica

1. Punteros

Direcciones y referencias (1)

- Cuando se declara una variable se asocian tres atributos fundamentales:
 - **Nombre:** identificador de la variable
 - **Tipo de dato:** tipo de dato que almacenará la variable
 - **Dirección:** dirección de memoria donde se almacenará la variable

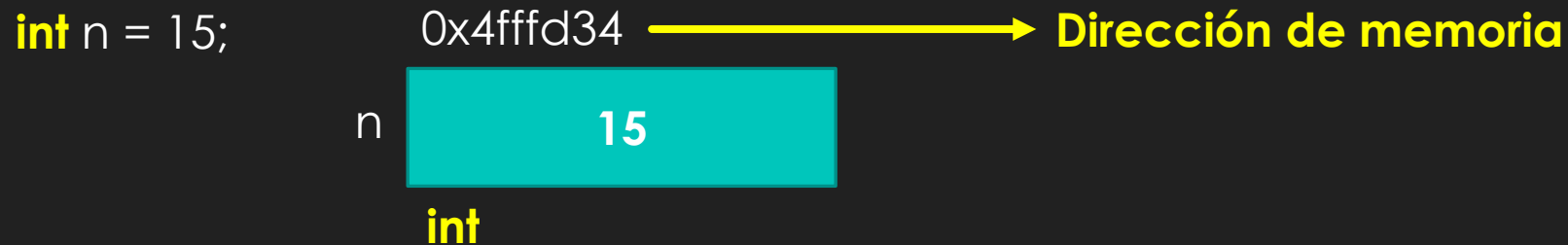
`int n = 15;`

0x4fffd34 → **Dirección de memoria**

n

int

15



1. Punteros

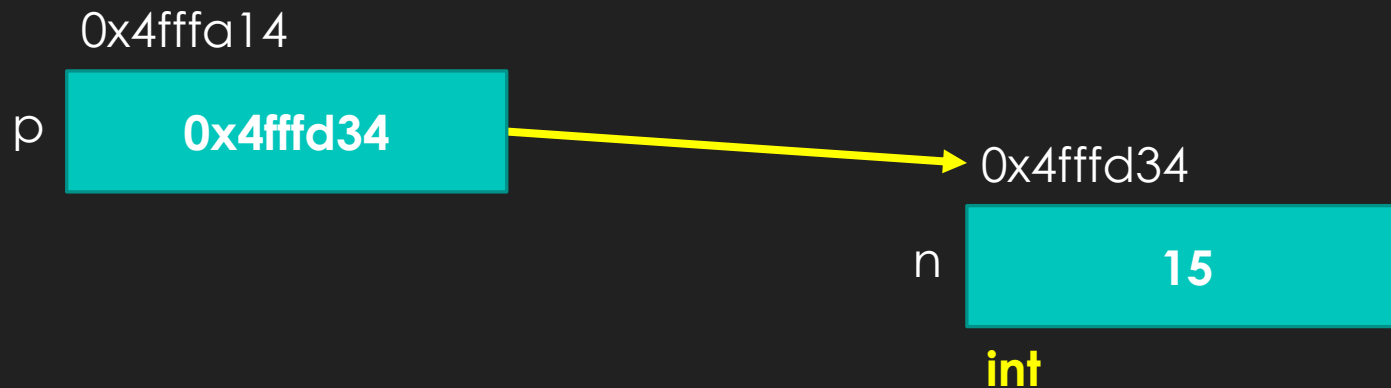
Direcciones y referencias (2)

- Al valor de una variable se accede mediante su nombre:
 - **Ejemplo:** `cout << n;`
- A la dirección de la variable se accede mediante su dirección (operador &):
 - **Ejemplo:** `cout << &n;`

1. Punteros

Concepto de puntero

- Un puntero es una **dirección** de memoria
- Una variable de tipo puntero contiene una dirección que apunta a otra posición de memoria
- Un puntero, en definitiva, apunta a otra variable de la memoria



1. Punteros

Declaración de un puntero

- Las variables de tipo puntero deben ser declaradas antes de utilizarlas
- Tenemos que indicar al compilador el **tipo de dato** al que apunta el puntero
- El operador **asterisco** (*) se utiliza para indicar que la variable es de tipo puntero

tipo_dato * identificador

int * puntero;

1. Punteros

Inicialización de un puntero

- Tenemos que **inicializar** la variable de tipo puntero antes de poder utilizarla
- La inicialización proporciona al puntero la dirección del dato correspondiente
- Podemos almacenar en un puntero la dirección de otra variable usando **&** (ampersand)
- Los **tipos de datos** de la variable puntero y la variable apuntada deben coincidir

- **Ejemplo:**

```
int i = 15;  
int *p;  
p = &i;
```

1. Punteros

Indirección de punteros

- Podemos **navegar** a la dirección de memoria que apunta un puntero con el operador *****

Dirección de memoria

0x4fffd35

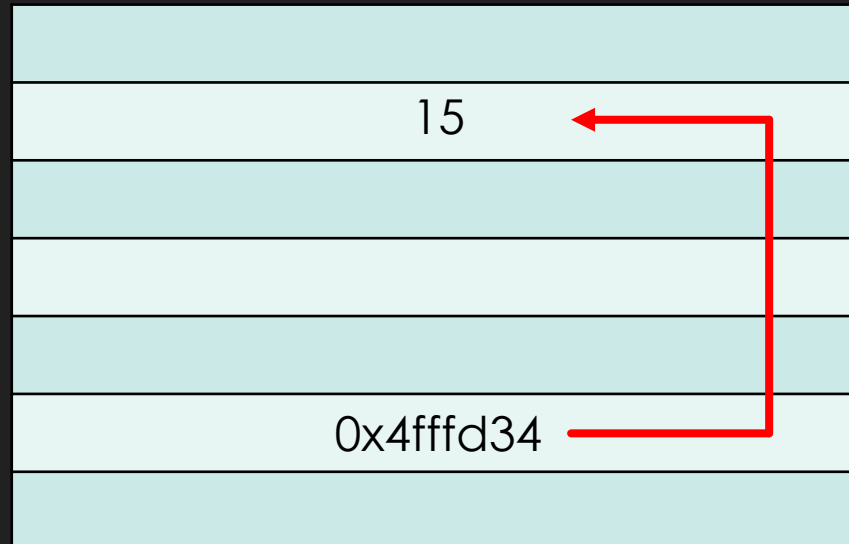
0x4fffd34

...

0x4fffd10

0x4fffd09

0x4fffd08



int n = 15;

cout << *p;

Salida: 15

cout << &n;

Salida: 0x4fffd34

int *p = &n;

cout << p;

Salida: 0x4fffd34

1. Punteros

Puntero nulo

- Un puntero nulo no apunta a ninguna variable
- No contiene ninguna dirección de memoria

- **Ejemplo:**

```
int *p = NULL;
```

1. Punteros

Puntero void

- Un puntero void puede direccionar **cualquier dirección de memoria**
- No está asociado a ningún tipo de dato concreto
- No confundir valor **NULL** (valor nulo) con **void** (void es un tipo de dato)
- **Ejemplo:**

```
int n = 15;  
void *p = &n;
```

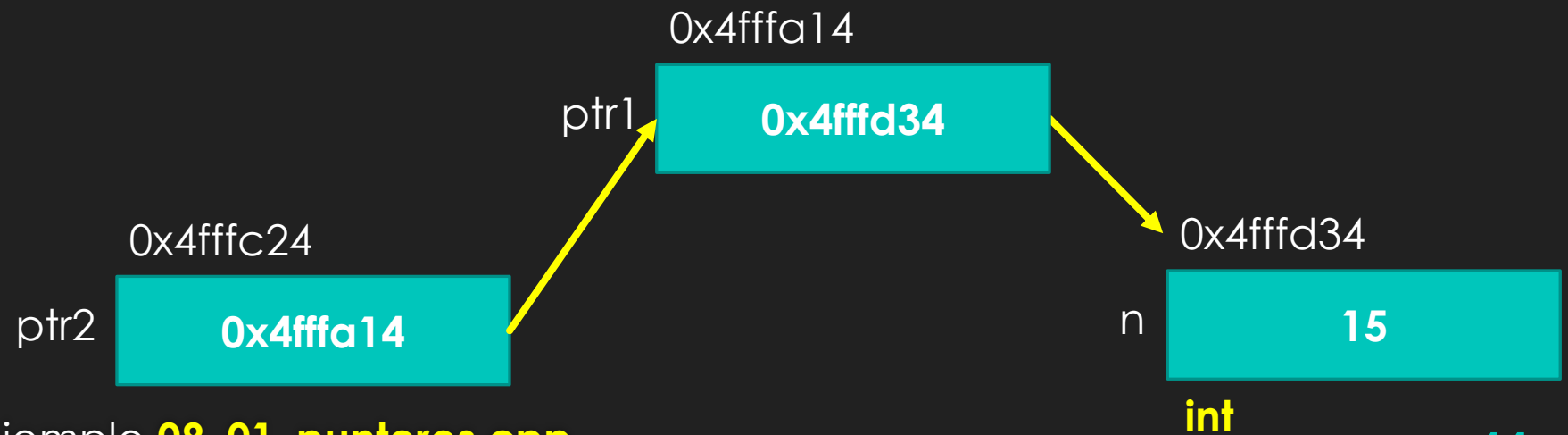
1. Punteros

Puntero a puntero

- Un puntero puede apuntar a otra variable puntero
- Para declarar un puntero a puntero se usan dos asteriscos (**)

- **Ejemplo:**

```
int n = 15;  
int *ptr1 = &n;  
int **ptr2 = &ptr1;
```



- Abre el fichero de ejemplo **08_01_punteros.cpp**

1. Punteros

Punteros y arrays

- Los arrays y los punteros están fuertemente relacionados
- Internamente, un array es tratado como un puntero (el nombre del array es un puntero)



1. Punteros

Aritmética de punteros

- Un puntero es una **dirección**, por lo que solo tienen sentido **ciertas** operaciones
- **Operaciones no válidas:**
 - No se pueden sumar dos punteros
 - No se pueden multiplicar dos punteros
 - No se pueden dividir dos punteros

1. Punteros

Paso por valor

- En el **paso por valor**, los argumentos que recibe la función son una copia
- **Los argumentos tienen como ámbito el bloque de código de la función**
- Fuera de la función, los argumentos podrían tomar su valor original
- Abre el fichero de ejemplo **08_02_paso_valor.cpp**

1. Punteros

Paso por referencia (usando punteros)

- En el **paso por referencia**, la función recibe las direcciones de memoria de las variables
- **Es similar a las variables globales, pero con mucho más control y depuración**
- Fuera de la función, los argumentos han sido modificados
- Abre el fichero de ejemplo **08_03_paso_referencia.cpp**

1. Punteros

Punteros a estructuras

- Un puntero también puede apuntar a una estructura
- La única excepción que hay que tener en cuenta es a la hora de acceder a los elementos
- Para acceder a cada elemento de la estructura a través del puntero se emplea ->
- **Ejemplo:**
Persona persona;
Persona *ptr = &persona;
cout << ptr->nombre;
- Abre el fichero de ejemplo **08_04_punteros_estructuras.cpp**

2. Memoria dinámica

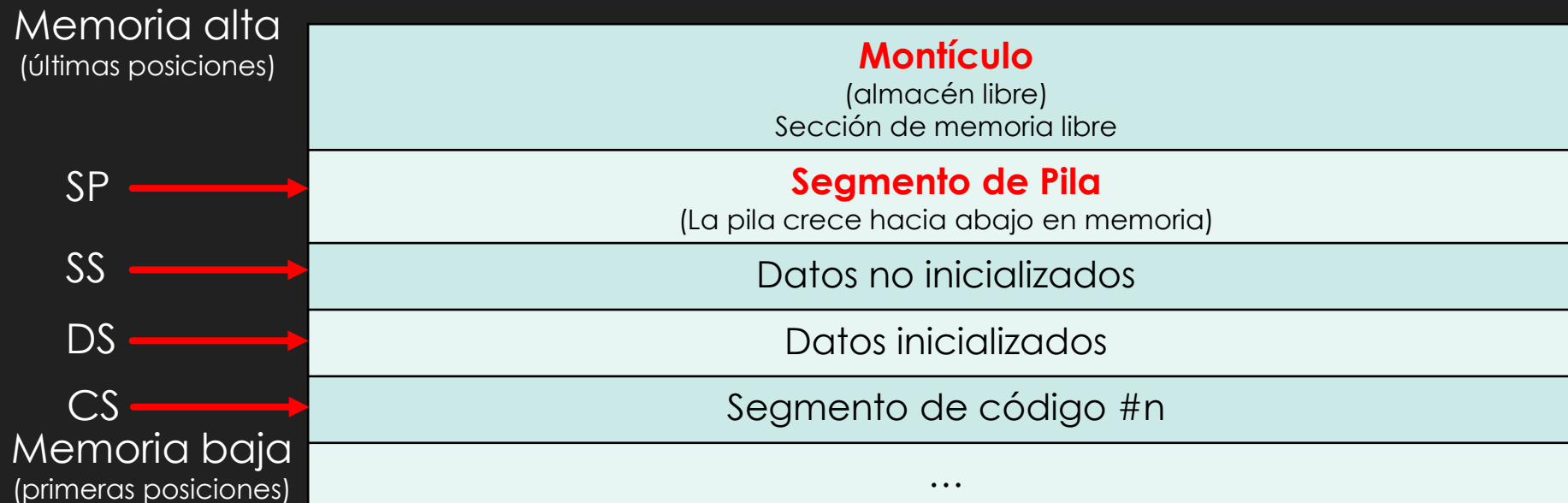
Gestión dinámica de la memoria

- La memoria principal del ordenador se divide en: **memoria estática** y **memoria dinámica**
- La memoria estática la ocupan las variables declaradas estáticamente
- La memoria dinámica se ocupa y libera mediante el uso de los operadores **new** y **delete**
- C++ carece de un **recolector de basura** automático (garbage collector), como **Java**
- La memoria dinámica debe **liberarse** obligatoriamente usando **delete**
- La memoria dinámica permite crear estructuras de datos sin saber el tamaño al inicio

2. Memoria dinámica

Mapa de memoria de un programa

- El mapa de memoria de un programa se asemeja al de la siguiente figura:
- Cada segmento suele limitarse a 64k (suele depender de cada arquitectura):



2. Memoria dinámica

Operador new (1)

- El operador **new** asigna un **bloque de memoria** a una variable (en función del tipo de dato)
- El operador new devuelve como parámetro un **puntero** al bloque de memoria asignado
- Los tipos de datos **deben coincidir**
- **Ejemplo:**

```
int * ptr;
```



```
ptr = new int;
```

 Palabra reservada **new**

2. Memoria dinámica

Operador new (2)

- El operador **new** asigna un **bloque de memoria** a una variable (en función del tipo de dato)
- El operador new devuelve como parámetro un **puntero** al bloque de memoria asignado
- Los tipos de datos **deben coincidir**
- **Ejemplo (vector):**

int * ptr;

↑ Longitud (número de variables)

ptr = **new int** [5];

↓ Palabra reservada **new**

2. Memoria dinámica

Operador new (3)

- **Precauciones:**
 - El almacenamiento dinámico no es una fuente inagotable de memoria
 - Si al ejecutar **new** no hay memoria, se devolverá **NULL** (¡hay que comprobarlo siempre!)
 - Es importante liberar la memoria cuando no la vayamos a utilizar (operador delete)

2. Memoria dinámica

Operador delete

- Se encarga de liberar la memoria dinámica que ha sido reservada con new

- **Ejemplo:**

```
int * ptr;  
ptr = new int;  
*ptr = 5;  
delete ptr;
```

2. Memoria dinámica

Arrays dinámicos

- Es muy usual recurrir a los punteros y a la memoria dinámica para definir arrays dinámicos
- El **array dinámico** tendrá el número de elementos que indiquemos en tiempo de ejecución
- El borrado del array dinámico se realiza con la instrucción **delete [] array;**
- Abre los ficheros **08_05_arrays_dinamicos.cpp** y **08_06_arrays_bidimensionales_dinamicos.cpp**

2. Memoria dinámica

Arrays de estructuras dinámicos

- También es posible crear un array de estructuras de manera dinámica
- La creación se lleva a cabo como igual que en el caso de un array dinámico
- Abre el fichero de ejemplo **08_07_arrays_estructuras_dinamicos.cpp**

Créditos de las imágenes y figuras

Cliparts e iconos

- **Obtenidos mediante la herramienta web [IconFinder](#)** (según sus disposiciones):
 - Diapositiva 1
 - Según la plataforma IconFinder, dicho material puede usarse libremente (free comercial use)
 - A fecha de edición de este material, todos los cliparts son free for comercial use (sin restricciones)

Resto de diagramas y gráficas

- Se han desarrollado en PowerPoint y se han incrustado en esta presentación
- Todos estos materiales se han desarrollado por el autor
 - Si se ha empleado algún icono externo, este se rige según lo expresado anteriormente